APPENDIX 1

```
pure function calc_crc(
  constant polynomial : std_logic_vector;
    --the polynomial represented as a vector
    --Eg. x^4 + x^2 + 1x^0 => polynomial := " 10101"
    --Eg HDLC-16 CRC Poly is x^16 + x^12 + x^5 + 1 => "10001000000100001";
    --Eg HDLC-32 CRC Poly is
    --  x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x^1+x^0
    --  => "100000100110000010001110110110111";
  constant augmented_message : boolean;
    --when true, the message must be augmented
    --  (followed by 'n' bits where 'n' = bits in crc)
    --when false, the message does not have to be augmented.
  crc_in : std_logic_vector;
    --current value of the crc MUST have MSB in the LEFTMOST bit.
  data_in : std_logic_vector
    --The new data word. This is always processed LEFT to RIGHT
  ) return std_logic_vector is
    -- returned CRC value will always have MSB in the LEFTMOST bit.

    constant data_bits : integer := data_in'length;
    constant crc_bits : integer := crc_in'length;
    variable data : std_logic_vector(data_bits-1 downto 0);
    variable crc_new : std_logic_vector(crc_bits-1 downto 0);
    variable crc : std_logic_vector(crc_bits-1 downto 0);
    variable crc_out : std_logic_vector(crc_in'range);

    variable c_polynomial : std_logic_vector(crc_bits downto 0) := polynomial;
      --avoids problems with range direction
begin
  data := data_in; --avoids problems with range direction
  crc := crc_in;
  for word_bit_num in data_bits-1 downto 0 loop
    if augmented_message then
      --this algorithm requires augmented message
      --(ie message appended with 'n' zero's where 'n' is num of bits in CRC)
      for crc_bit_num in crc_bits-1 downto 1 loop
        if c_polynomial(crc_bit_num) = '1' then
          crc_new(crc_bit_num) := crc(crc_bit_num-1) xor crc(crc_bits-1);
        else
          crc_new(crc_bit_num) := crc(crc_bit_num-1);
        end if;
      end loop;
      crc_new(0) := data(word_bit_num) xor crc(crc_bits-1);
      crc := crc_new;
    else
      --this algorithm does not require an augmented message
      for crc_bit_num in crc_bits-1 downto 1 loop
        if c_polynomial(crc_bit_num) = '1' then
          crc_new(crc_bit_num) := crc(crc_bit_num-1) xor (crc(crc_bits-1) xor
data(word_bit_num));
        else
          crc_new(crc_bit_num) := crc(crc_bit_num-1);
        end if;
      end loop;
      crc_new(0) := data(word_bit_num) xor crc(crc_bits-1);
      crc := crc_new;
    end if; --augmented message
  end loop;
  crc_out := crc_new; --match output to port range
  return crc_out;

end function calc_crc;
```

APPENDIX 2

```
    pure function identify_data_terms(
            augmented_message : boolean;
            crc_bits : integer;
            polynomial : std_logic_vector;
            data_bits : integer
        ) return data_xor_ena_vec_array_type is

        variable data_xor_ena_vec : data_xor_ena_vec_array_type;
        variable data_bit_enable : std_logic_vector(data_bits-1 downto 0);
        variable data_bit_xor_ena : std_logic_vector(crc_bits-1 downto 0);
        constant zero_crc : std_logic_vector(crc_bits-1 downto 0) := (others
=> '0');

    begin
        data_xor_ena_vec := (others => (others => '0'));
        for word_bit_num in data_bits-1 downto 0 loop
            --create a mask to represent the current word_bit_num
            --eg 100000000 or 01000000 etc
            data_bit_enable := (others => '0');
            data_bit_enable(word_bit_num) := '1';
            --find out if this data bit is a term
            --in each crc bit's xor function
            data_bit_xor_ena := calc_crc(
                                    polynomial => polynomial,
                                    augmented_message => augmented_message,
                                    crc_in => zero_crc,
                                    data_in => data_bit_enable
                                );
            --assign the result to the appropriate entry in the
data_xor_ena_vec
            for crc_bit in crc_bits-1 downto 0 loop
                data_xor_ena_vec(crc_bit)(word_bit_num) :=
data_bit_xor_ena(crc_bit);
            end loop;

        end loop;

        return data_xor_ena_vec;

    end function identify_data_terms;
```

APPENDIX 3

```
pure function identify_crc_terms(
        augmented_message : boolean;
        crc_bits : integer;
        polynomial : std_logic_vector;
        data_bits : integer
    ) return crc_xor_ena_vec_array_type is

    variable crc_xor_ena_vec : crc_xor_ena_vec_array_type;
    variable crc_bit_enable : std_logic_vector(crc_bits-1 downto 0);
    variable crc_bit_xor_ena : std_logic_vector(crc_bits-1 downto 0);
    constant zero_data : std_logic_vector(data_bits-1 downto 0) := (others
=> '0');
    variable c_polynomial : std_logic_vector(crc_bits downto 0) :=
polynomial;
        --avoids direction ambiguity

begin
    crc_xor_ena_vec := (others => (others => '0'));
    for word_bit_num in crc_bits-1 downto 0 loop
        --create a mask to represent the current word_bit_num
        --eg 100000000 or 01000000 etc
        crc_bit_enable := (others => '0');
        crc_bit_enable(word_bit_num) := '1';
        --find out if this crc bit is a term
        --in each crc bit's xor function
        crc_bit_xor_ena := calc_crc(
                                polynomial => c_polynomial,
                                augmented_message => augmented_message,
                                crc_in => crc_bit_enable,
                                data_in => zero_data
                            );
        --assign the result to the appropriate entry in the
crc_xor_ena_vec
        for crc_bit in crc_bits-1 downto 0 loop
            crc_xor_ena_vec(crc_bit)(word_bit_num) :=
crc_bit_xor_ena(crc_bit);
        end loop;

    end loop;

    return crc_xor_ena_vec;

end function identify_crc_terms;
```